Java Type System – Proposals for Java 10 or 11

Martin Plümicke

Baden-Wuerttemberg Cooperative State University
Stuttgart/Horb

5. Oktober 2015

Proposals for extentions:

1. Real function types
2. Type–Inference
3. Generices in Byte-Code
4. Principal Type

1. Real function types

Problems as Java has no real function types

- Lambda–expressions have no explicit type
  (Functional interface are target types)
- Subtyping of functional interfaces can only be simulated by wildcards
- Direct application of Lambda–expressions to arguments can only be done by type-casts
  (Lambda–expressions have no explicite types.)

Introduction of function types in the Scala–style

```
interface FunN*<+R,-T1, ..., -TN> {
    R apply(T1 arg1, ..., TN argN);
}
```

where

- $\texttt{Fun}N*\texttt{<}\texttt{T}_0, \texttt{T}'_1, \ldots, \texttt{T}'_N\texttt{>} \leq^* \texttt{Fun}N*\texttt{<}\texttt{T}'_0, \texttt{T}_1, \ldots, \texttt{T}_N\texttt{>}$ iff $\texttt{T}_i \leq^* \texttt{T}'_i$
- For $\texttt{Fun}N*$ no wildcards are allowed.

Proposal: Lambda–expressions are explicitly typed by FunN*–types

Example: `x -> y -> f -> f.apply(x,y)`

Function type: `A -> (B -> (((A, B) -> C) -> C)))`

Example: `x -> y -> f -> f.apply(x,y)`

Function type: `A -> (B -> (((A, B) -> C) -> C)))`

Java 8–Type:

```
Fun1<? extends Fun1<? extends Fun1<? extends C,
                                   ? super Fun2<? extends C,
                                                ? super A,
                                                ? super B>>,
                            ? super B>,
        ? super A>
```

Example: `x -> y -> f -> f.apply(x,y)`

Function type: `A -> (B -> (((A, B) -> C) -> C)))`

Java 8–Type:

```
Fun1<? extends Fun1<? extends Fun1<? extends C,
                                   ? super Fun2<? extends C,
                                                ? super A,
                                                ? super B>>,
                            ? super B>,
        ? super A>
```

Function type in Scala–style:

```
Fun1*<Fun1*<Fun1*<C,Fun2*<C,A,B>>,B>,A>
```

2. Type–Inference

$$\mathbf{TI}\colon \texttt{TypeAssumptions} \times \texttt{Class} \to \{\, (\texttt{Constraints}, \texttt{TClass}) \,\}$$

$$\mathbf{TI}(\, \mathit{Ass}, \mathrm{Class}(\, \tau, \mathrm{extends}(\, \tau' \,), \mathit{fdecls} \,) \,) =$$

let

$$\frac{(\mathrm{Class}(\, \tau, \mathrm{extends}(\, \tau' \,), \mathit{fdecls}_t \,), \mathit{ConS})}{\mathbf{TYPE}(\, \mathit{Ass}, \mathrm{Class}(\, \tau, \mathrm{extends}(\, \tau' \,), \mathit{fdecls} \,) \,)}$$

$$\frac{\{\, (cs_1, \sigma_1), \ldots, (cs_n, \sigma_n) \,\}}{} = \mathbf{SOLVE}(\, \mathit{ConS} \,) \quad \rightsquigarrow (\mathit{Type} - \mathit{Unification})$$

in

$$\{\, (cs_i, \sigma_i(\, \mathrm{Class}(\, \tau, \mathrm{extends}(\, \tau' \,), \mathit{fdecls}_t \,) \,)) \mid 1 \leqslant i \leqslant n \,\}$$

Type–Unify

Type–Unify: $\texttt{Constraints} \rightarrow \{\, \texttt{Substitutions} \,\} \cup \{\, \mathit{fail} \,\}$

Input: $\{\, (\theta_1 \lessdot \theta'_1), \ldots, (\theta_n \lessdot \theta'_n) \,\}$

Output: $\{\, \sigma_1, \ldots, \sigma_m \,\}$

Post-condition: $\forall_j \{\, (\sigma_j(\,\theta_1\,) \leq^* \sigma_j(\,\theta'_1\,)), \ldots, (\sigma_j(\,\theta_n\,) \leq^* \sigma_j(\,\theta'_n\,)) \,\}$

where $\leq^*$ is the sub-typing relation

Type-Unification extentions

$$(\text{redFun}N^*)\ \frac{Eq \cup \{\, \texttt{Fun}N*\!<\!\theta, \theta'_1, \ldots, \theta'_N\!> \lessdot \texttt{Fun}N*\!<\!\theta', \theta_1, \ldots, \theta_N\!> \,\}}{Eq \cup \{\, \theta \lessdot \theta', \theta_1 \lessdot \theta'_1, \ldots, \theta_N \lessdot \theta'_N \,\}}$$

$$(\text{grFun}N^*)\ \frac{Eq \cup \{\, \texttt{Fun}N*\!<\!\theta, \theta'_1, \ldots, \theta'_N\!> \lessdot a \,\}}{Eq \cup \{\, a \doteq \texttt{Fun}N*\!<\!b', b_1, \ldots, b_N\!>, \theta \lessdot b', b_i \lessdot \theta'_i \,\}}\ b', b_i \text{ fresh}$$

$$(\text{smFun}N^*)\ \frac{Eq \cup \{\, a \lessdot \texttt{Fun}N*\!<\!\theta', \theta_1, \ldots, \theta_N\!> \,\}}{Eq \cup \{\, a \doteq \texttt{Fun}N*\!<\!b, b'_1, \ldots, b'_N\!>, b \lessdot \theta', \theta_i \lessdot b'_i \,\}}\ b, b_i \text{ fresh}$$

Example

```
class Matrix extends Vector<Vector<Integer>> {

    op = (m) -> (f) -> f.apply(this, m);
}
```

Example

```
class Matrix extends Vector<Vector<Integer>> {
    <y2', b1 extends y2', d', y extends d'>
    Fun1*<Fun1*<d', Fun2*<y, X, y2'>>, b1>
    op = (m) -> (f) -> f.apply(this, m);
}
```

$\texttt{Matrix} \leq^* X$

3. Generices in Byte-Code

Function types and type-erasures

Overloading example

```
void apply(Fun*1<Integer, Integer> f) { ... }

void apply(Fun*1<Boolean, Boolean> f) { ... }
```

3. Generices in Byte-Code

Function types and type-erasures

Overloading example

```
void apply(Fun*1<Integer, Integer> f) { ... }

void apply(Fun*1<Boolean, Boolean> f) { ... }
```

Leads in byte-code to (type-erasure):

```
void apply(Fun*1 f) { ... }

void apply(Fun*1 f) { ... }
```

Ambigous overloading!

Example: Scalarproduct

Program without type declaration

```
scalar_product(x,y) {

    while (i < x.size()) {
        ret = ret + x.elementAt(i) * y.elementAt(i);
    }
    return ret;
}
```

Example: Scalarproduct

Typing in Byte-Code

```
Integer scalar_product(Vector<Integer> x, Vector<Integer> y) {

    while (i < x.size()) {
        ret = ret + x.elementAt(i) * y.elementAt(i);
    }
    return ret;
}

Double scalar_product(Vector<Double> x, Vector<Double> y) {

    while (i < x.size()) {
        ret = ret + x.elementAt(i) * y.elementAt(i);
    }
    return ret;
}
```

$\Rightarrow$ **Ambigous overloading!!!**

4. Principal Type

Definition

An expression $\texttt{exp}$ is typeable if and only if there is a pair (Ass, ty) of a set of assumptions and a type, called the principal pair for $\texttt{exp}$, such that:

1. $Ass \vdash \texttt{exp} : ty$, and
2. for every pair (Ass', ty'), such that $Ass' \vdash \texttt{exp} : ty'$ there exists an operation O such that $O(\,(Ass, ty)\,) = (Ass', ty')$.

Type of op (determinded by **TI**):

```
Fun1*<Fun1*<d′,Fun2*<y,Vector<? ext. Vector<? ext. Integer>>, y2′>>,b1>
&
Fun1*<Fun1*<d′,Fun2*<y,Vector<? ext. Vector<? super Integer>>, y2′>>,b1>
&
Fun1*<Fun1*<d′,Fun2*<y,Vector<? extends Vector<Integer>>, y2′>>,b1>
&
Fun1*<Fun1*<d′,Fun2*<y,Vector<? super Vector<Integer>>, y2′>>,b1>

& ... &

Fun1*<Fun1*<d′,Fun2*<y,Vector<Vector<Integer>>, y2′>>,b1>
&
Fun1*<Fun1*<d′,Fun2*<y,Matrix, y2′>>,b1>
```

Reduced type of op

```
Fun1*<Fun1*<d′,Fun2*<y,Vector<? ext. Vector<? ext. Integer>>,y2′>>,b1>
&
Fun1*<Fun1*<d′,Fun2*<y,Vector<? ext. Vector<? super Integer>>,y2′>>,b1>
&
Fun1*<Fun1*<d′,Fun2*<y,Vector<? super Vector<Integer>>,y2′>>,b1>
```

Principal Type

An intersection type with minimal number of elements of an expression is a principal type, if any (non-intersection) type of the expression is a subtype of a generic instance of one element of the intersection type (and the call-graphs are identical).

Operation O: Projection and generic instance.

Conclusion and Outlook

Conclusion

1. Real Function Types
2. Type Inference
3. Generices in Byte-Code
4. Principal Type

Conclusion and Outlook

Conclusion

1. Real Function Types
2. Type Inference
3. Generices in Byte-Code
4. Principal Type

Outlook

- Bytecode without type-erasures
- Type–Inference with abstract fields and methods
- Implementation

Stellenausschreibung

DHBW (M. Plümicke) gemeinsam mit der Uni Freiburg (P. Thiemann)

Wissenschaftlicher Mitarbeiter/in

Promotionsthema: Real function types in Java

Bei Interesse: Martin Plümicke, pl@dhbw.de, 07451-521142